

Orchard| Mindtree



Mindtree

*Welcome to possible*

Jasmine and Karma

May 17, 2018

By

Mujahid Islam (M104457)

## What is Jasmine?

Jasmine is a java script testing framework that supports a software development practice called [Behaviour Driven Development](#), or BDD for short. It's a specific flavour of [Test Driven Development](#) (TDD).

Jasmine, and BDD in general, attempts to describe tests in a human readable format so that non-technical people can understand what is being tested. However even if you *are* technical reading tests in BDD format makes it a lot easier to understand what's going on.

We write test code in jasmine and it allow us to structure our test code.

For example if we wanted to test this function:

```
function helloWorld() {  
    return 'Hello world!';  
}
```

We would write a jasmine test *spec* like so:

```
describe('Hello world', () => {  
    it('says hello', () => {  
        expect(helloWorld())  
            .toEqual('Hello world!');  
    });  
});
```

## Setup and Teardown:

Sometimes in order to test a feature we need to perform some setup, perhaps it's creating some test objects. Also we may need to perform some clean-up activities after we have finished testing, perhaps we need to delete some files from the hard drive.

These activities are called *setup* and *teardown* (for cleaning up) and Jasmine has a few functions we can use to make this easier:

### **beforeAll**

This function is called **once**, *before* all the specs in `describe` test suite are run.

### **afterAll**

This function is called **once** *after* all the specs in a test suite are finished.

### **beforeEach**

This function is called *before* **each** test specification, `it` function, has been run.

### **afterEach**

This function is called *after* **each** test specification has been run.

```
describe('Hello world', () => {

  let expected = "";

  beforeEach(() => {
    expected = "Hello World!";
  });

  afterEach(() => {
    expected = "";
  });

  it('says hello', () => {
    expect(helloWorld())
      .toEqual(expected);
  });
});
```

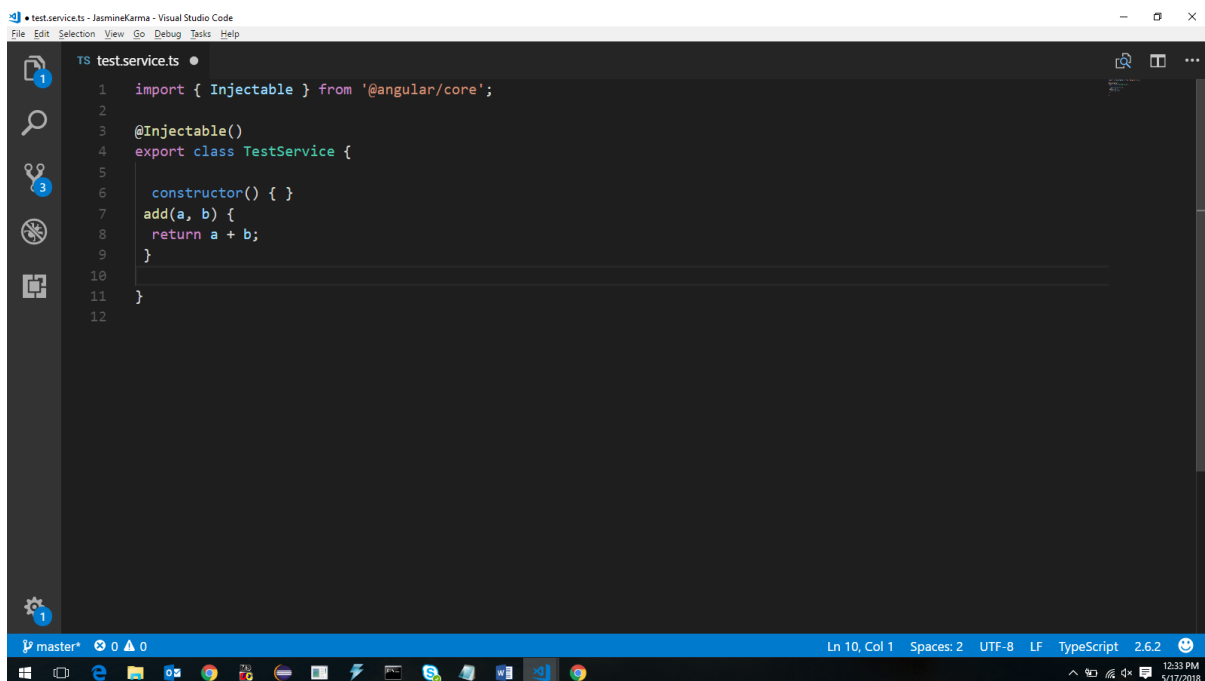
Above steps are similar for all type of component.

For Services:

Suppose I have created one service named 'test' by running a command "ng g s test".

Then two files will be created one is [test.service.ts](#) and another one is [test.service.spec.ts](#).

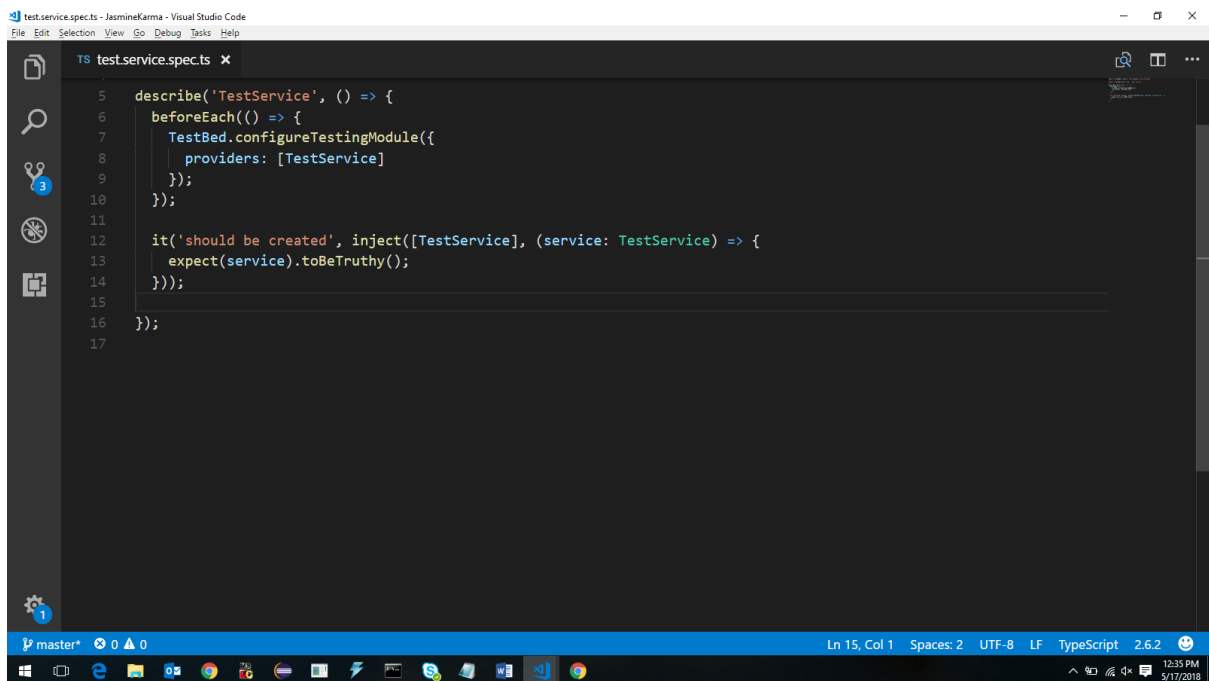
In [test.service.ts](#) file we have method that return the sum of two number as show in below image.



```
1  import { Injectable } from '@angular/core';
2
3  @Injectable()
4  export class TestService {
5
6      constructor() { }
7      add(a, b) {
8          return a + b;
9      }
10
11  }
12
```

Now, we will write test cases for [test.service.ts](#) and one function that is present there.

Initially the test.service.spec.ts will look like:

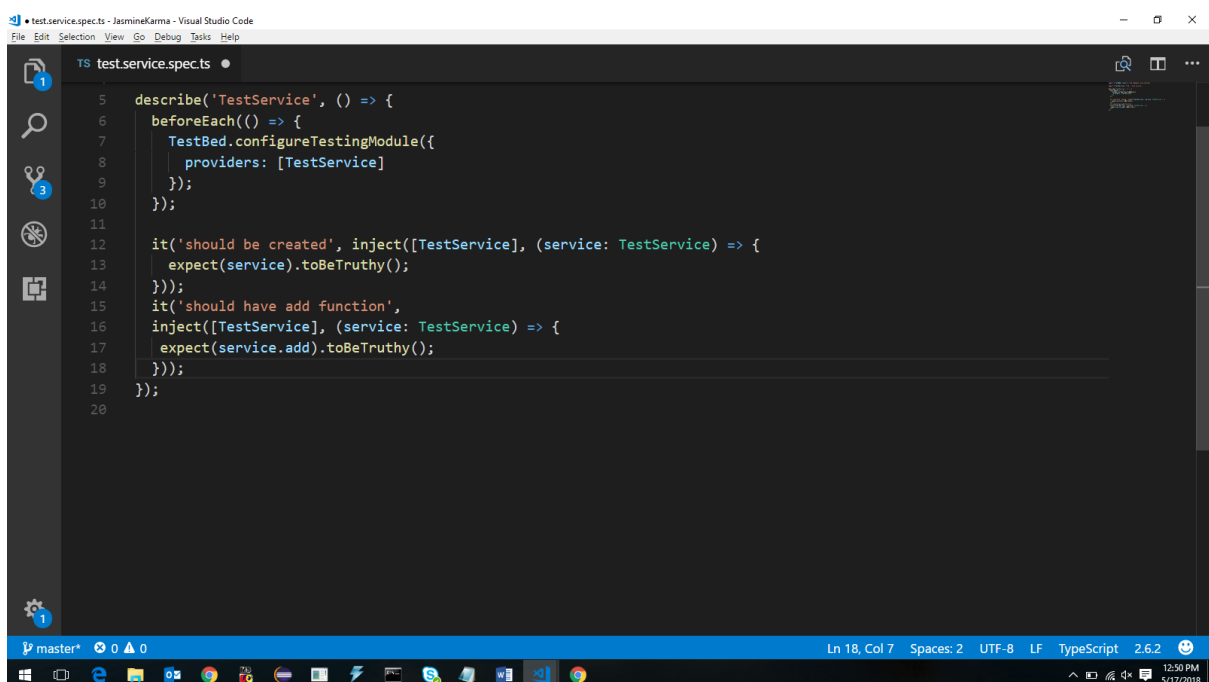


The screenshot shows a Visual Studio Code editor window with the file 'test.service.spec.ts' open. The code is written in TypeScript and uses Jasmine for testing. It defines a 'TestService' and a 'beforeEach' block that configures the testing module with 'TestService' as a provider. A single test 'should be created' is shown, which injects 'TestService' and expects it to be truthy.

```
5 describe('TestService', () => {
6   beforeEach(() => {
7     TestBed.configureTestingModule({
8       providers: [TestService]
9     });
10  });
11
12  it('should be created', inject([TestService], (service: TestService) => {
13    expect(service).toBeTruthy();
14  }));
15
16 });
17
```

First of we have to check whether our “add()” function is present in our service or not.

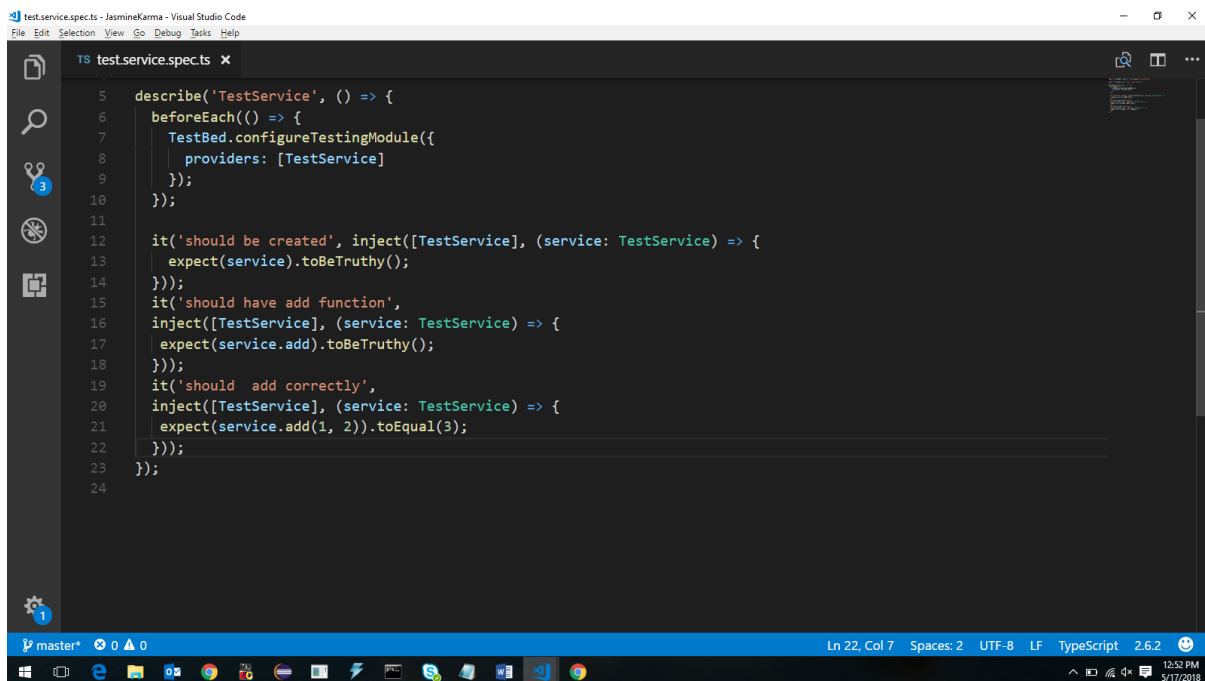
So for this we use “toBeTruthy()” function.



The screenshot shows the same Visual Studio Code editor window, but the test file has been updated. A second test 'should have add function' has been added, which injects 'TestService' and expects the 'add' property to be truthy.

```
5 describe('TestService', () => {
6   beforeEach(() => {
7     TestBed.configureTestingModule({
8       providers: [TestService]
9     });
10  });
11
12  it('should be created', inject([TestService], (service: TestService) => {
13    expect(service).toBeTruthy();
14  }));
15  it('should have add function',
16    inject([TestService], (service: TestService) => {
17      expect(service.add).toBeTruthy();
18    }));
19 });
20
```

After that we have to write a test case for our “add()” function.



```
5 describe('TestService', () => {
6   beforeEach(() => {
7     TestBed.configureTestingModule({
8       providers: [TestService]
9     });
10  });
11
12  it('should be created', inject([TestService], (service: TestService) => {
13    expect(service).toBeTruthy();
14  }));
15  it('should have add function',
16    inject([TestService], (service: TestService) => {
17      expect(service.add).toBeTruthy();
18    }));
19  it('should add correctly',
20    inject([TestService], (service: TestService) => {
21      expect(service.add(1, 2)).toEqual(3);
22    }));
23 });
24
```

## What is Karma?

Manually running Jasmine tests by refreshing a browser tab repeatedly in different browsers every-time we edit some code can become tiresome.

Karma is a tool which lets us spawn browsers and run jasmine tests inside of them all from the command line. The results of the tests are also displayed on the command line.

Karma can also watch your development files for changes and re-run the tests automatically.

Karma lets us run jasmine tests as part of a development tool chain which requires tests to be runnable and results inspect able via the command line.

It's not necessary to know the internals of how Karma works. When using the Angular CLI it handles the configuration for us and for the rest of this section we are going to run the tests using only Jasmine. Simply, Karma allow us to run the test cases in different browser.

Now run “ng test” command and check all test cases are passing or not.

The below result will come. In this u can see all the spec are passing there will be a zero failure.

